

### **AMENDMENTS TO THE DRAWINGS**

Submitted herewith on separate sheets of paper are replacement Figures 1, 3A, 3B, 3C, 3D and 3E. Each has been changed to remove underling to element 100 and 300 and include an indicator arrow for each.

Attachment: Replacement Sheets (6)

## **REMARKS**

In response to the above identified Office Action, Applicants have amended the application and seek reconsideration thereof. In this response, Applicants do not add or cancel any claims. Claims 1-6 and 19 have been amended. Accordingly claims 1-21 remain pending in the application.

### **I. Objection to The Title**

The Examiner has objected to the title of the invention asserting that the title of the invention is not descriptive. The current title describes an embodiment of the claimed invention. It is unclear to the Applicants what further description of the claimed invention is needed in the title. If the Examiner maintains this objection, Applicants respectfully request that the Examiner clarify the basis for the objection. Specifically, Applicants request the Examiner set forth those elements of the claimed invention which the title fails to describe.

### **II. Objections to the Specification**

The Examiner has objected to the Specification asserting that it contains informalities. Specifically, the Examiner notes informalities in line 14 of paragraph 21 and line 11 of paragraph 28. Applicants have amended these paragraphs as requested by the Examiner. Accordingly, withdrawal of the objection to the Specification is requested.

### **III. Objections to the Claims**

The Examiner has objected to Claims 3, 6 and 19 asserting that each contains informalities. Applicants have amended each of these claims as requested by the Examiner. Accordingly, reconsideration and withdrawal of the objections to these claims are requested.

### **IV. Claims Rejected Under 35 U.S.C. §102**

Claims 1-21 stand rejected under 35 USC section §102 as being anticipated by U.S. Patent No. 6,807,621 issued to Strombergson et al. (hereinafter “Strombergson.”)

To anticipate a claim, a single reference must disclose each element of that claim. In regard to claim 1, this claim includes the elements of “a third device... to track relative segment order between the first device and the second device.” Applicants believe that Strombergson does not teach these elements of claim 1. The Examiner cites commit stage 5 in Fig. 1 as teaching this element of claim 1 and asserts “that all three devices track sequential data, as all are parts of a pipelined processor which fetches instructions in a sequence.” The Examiner’s assertion appears to be an acknowledgement that Strombergson is silent as to how program order is maintained. The assertion is based on information extrinsic to the cited reference upon which the anticipation reference is based. Thus, Strombergson does not teach each of the elements of claim 1. The Examiner makes similar assertions related to several other claims which have been rejected as anticipated by Strombergson. In each case, it appears that the Examiner is relying on information extrinsic to the cited reference to establish anticipation of these claims. Thus, in each instance where the Examiner relied on such an assertion Strombergson fails to teach of the elements of these claims and is not a proper basis for an anticipation rejection.

Also, the Examiner's assertions appear to indicate that he is operating under a misunderstanding as to the manner in which a standard reorder buffer functions. A standard reorder buffer does not track the relative order of instructions that have been assigned to different execution units. Rather, a standard reorder buffer is a queue in which the entire sequence of the instructions is placed that are being operated on by all execution units. As the execution units complete their execution of instructions the results are placed in the appropriate slot of the queue in the standard reorder buffer. See *High Performance Micro Processors: Chapter 8* at [www.cs.swan.ac.uk/~csneal/hpm/reorder.html](http://www.cs.swan.ac.uk/~csneal/hpm/reorder.html) (and submitted herewith). The Examiner has not indicated and the Applicants have been unable to discern any part of Strombergson that teaches that the reorder buffer in the commit stage is anything other than a standard reorder buffer. Thus, Strombergson does not teach "a third device...to track relative segment order between the first device and the second device." Therefore, Strombergson does not teach each of the elements of claim 1. Accordingly, reconsideration and withdrawal of the anticipation rejection of claim 1 are requested.

Claims 2-5 depend from independent claim 1 and incorporate the limitations thereof. Thus, at least for the reasons mentioned above in regard to independent claim 1, these claims are not anticipated by Strombergson. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

In regard to independent claims 6, 12, 15 and 19, these claims include elements similar to those of independent claim 1 including "tracking program order of the first set of instructions relative to the second set of instructions in a global reorder buffer," "a global reorder buffer to track instruction order of instructions assigned to the first reorder buffer relative to the second reorder buffer," "tracking program order of the first set of instructions relative to the second set of instructions in a global tracking device" and "a means for directing program order of the first

set of instructions relative to the second of instructions and the global tracking device.” Thus, at least for the reasons mentioned above in regard to independent claim 1, Strombergson does not anticipate each of these claims. Accordingly, reconsideration and withdrawal of the anticipation rejection of these claims are requested.

In regard to claims 7-11, 13, 14, 16-18, 20 and 21 these claims depend from independent claims 6, 12, 15 and 19, respectively, and incorporate the limitations thereof. Thus, at least for the reasons mentioned above in regard to claims 6, 12, 15 and 19, these claims are not anticipated by Strombergson. In regard to claim 17, this claim includes the element of “tracking a first set of switch points in a global tracking device.” As discussed above, the Examiner’s assertions regarding the functionality of a standard reorder buffer are incorrect. A standard reorder buffer is a queue data structure that has separate storage slots for each instruction and does not store a set of switch points. Therefore, Strombergson does not teach these elements of claim 17. Accordingly, reconsideration and withdrawal of the anticipation rejection of claims 7-11, 13, 14, 16-18, 20 and 21 are requested.

### CONCLUSION

In view of the foregoing, it is believed that all claims now pending, namely claims 1-21, patentably define the subject invention over the prior art of record, and are in condition for allowance and such action is earnestly solicited at the earliest possible date. If the Examiner believes that a telephone conference would be useful in moving the application forward to allowance, the Examiner is encouraged to contact the undersigned at (310) 207 3800.

Respectfully submitted,

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP

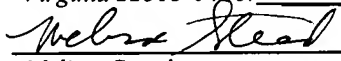
Dated: 2/24, 2006

  
Jonathan S. Miller Reg. No. 48,534

12400 Wilshire Boulevard, Seventh Floor  
Los Angeles, California 90025  
(310) 207-3800

### CERTIFICATE OF MAILING

*I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to:  
Commissioner of Patents, P.O. Box 1450, Alexandria, Virginia 22313-1450:*

  
Melissa Stead 2-24-06  
Date

[Previous](#) [Contents](#) [Next](#)

## Chapter 8: Reorder Buffers and Register Renaming

We have seen a range of possible ways that execution in a pipelined/superscalar machine could be delayed, but we have looked at actual mechanisms for reducing the effects of only some of them. For example, various strategies for *procedural dependency* (e.g. branch prediction, etc.), and have mentioned (if briefly) a strategy for resource conflict (duplication of resources). We have done nothing for those conflicts that involve data - i.e. true data dependency, output dependency and antidependency. We have not even made it clear, except informally, how we would go about recognizing their presence. In this chapter and the [next](#) we will start to address this. In particular, in this chapter, we will look at part of the solution to eliminating WAW and WAR hazards arising from output and antidependency, and mechanisms to enforce a precise architectural state. It turns out that the same hardware is involved in both of these steps.

As we saw [earlier](#) true data dependency is a property of a program, but name dependencies are not and can potentially be eliminated. Usually name dependencies involve conflicts in register use. One possible approach is to provide as many registers as possible, in an attempt to avoid clashes by giving the compiler plenty of choice. (This is basically the same solution used for resource conflicts - provide more resources.)

One problem is backward compatibility with existing architectures - these often have a limited number of registers, and there is no way to increase this number without radically redefining the architecture. Another problem is that a large register file means more work saving/restoring it when changing between processes (*context switching*).

### 8.1. Register Renaming

A devious strategy is *register renaming* - the hardware has a larg(ish) set of registers - often several times as many as the actual architecture claims to have. These registers are not associated permanently with the registers of the architecture, but are *dynamically* allocated as needed. Furthermore, there can be several versions of an architectural register present at any one time. Consider the following code:

```
MUL R2,R2,R3    ; R2 = R2 * R3
ADD R4,R2,1      ; R4 = R2 + 1
ADD R2,R3,1      ; R2 = R3 + 1
DIV R5,R2,R4     ; R5 = R4 * R4
```

Consider one problem: instruction 3 cannot go ahead until instruction 1 has finished, and instruction 2 has started. This is because there is an output dependency between instruction's 1 and 3 - both write to R2 - and an antidependency between instructions's 2 and 3 - instruction 3 overwrites instruction 2's argument. Now consider the same program, but with the registers labelled.

```
MUL R2_b,R2_a,R3_a
ADD R4_b,R2_b,1
```

```

ADD R2_c, R3_a, 1
DIV R5_b, R2_c, R4_b

```

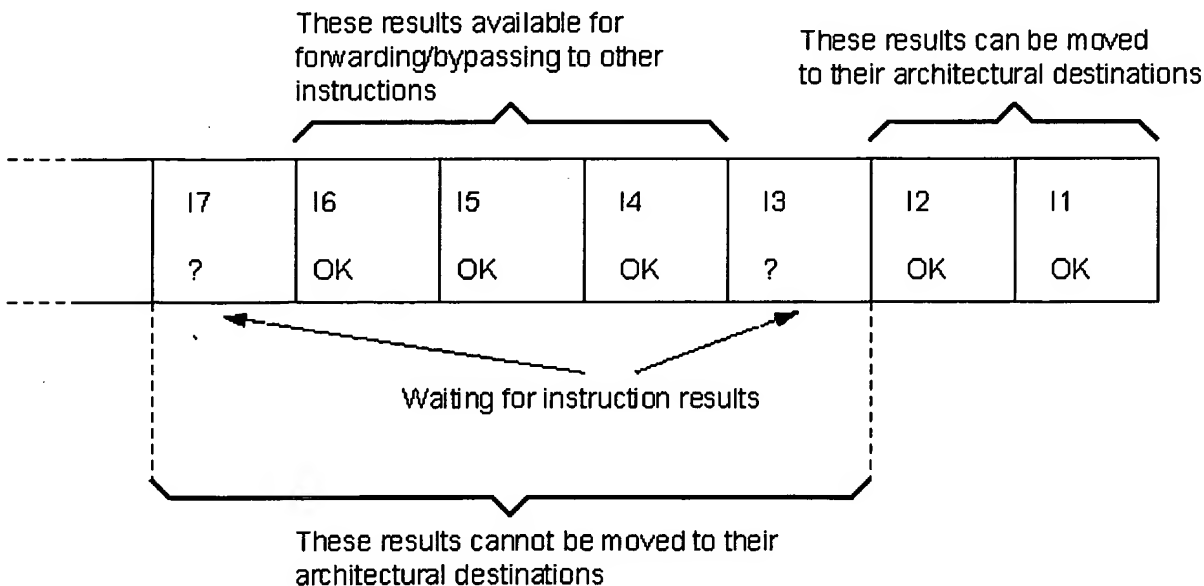
Now instruction 3 can start immediately, because it is using a 'different' R2 from instructions 1 and 2. We are effectively using a *history* of the contents of each register - for example, R2\_c is the newest version of R2, then R2\_b, followed by R2\_a (the oldest version).

There are two ways we can go about implementing register renaming. The first is by explicitly providing a larger set of registers than the architecture claims is present - a technique usually simply called register renaming. Alternatively (and more usually) by using a *reorder buffer*. We will look at reorder buffers first, and then briefly consider the alternative.

## 8.2. Reorder Buffers

Renaming based on a reorder buffer uses a physical register file that is the same size as the architectural register file, together with a set of registers arranged as a *queue* data structure, called the *reorder buffer*.

As instructions are issued, they are assigned entries for any results they may generate at the tail of the reorder buffer. That is, a place is reserved in the queue. We maintain the logical order of instructions within this buffer - so if we can issue four instructions  $i$  to  $i+3$  at once, we put  $i$  in the reorder buffer first, followed by  $i+1$ ,  $i+2$  and  $i+3$ . As instruction execution proceeds, the assigned entry will ultimately be filled in by a value, representing the result of the instruction. When entries reach the head of the reorder buffer, provided they've been filled in with their actual intended result, they are removed, and each value is written to its intended architectural register. If the value is not yet available, then we must wait until it is. Because instructions take variable times to execute, and because they may be executed out of program order, we may well find that the reorder buffer entry at the head of the queue is still waiting to be filled, while later entries are ready. In this case, all entries behind the unfilled slot must stay in the reorder buffer until the head instruction completes. For example, consider the case of 6 instructions I1 - I6. Suppose at a given clock cycle that I1 and I2 both finish, and that at earlier clock cycles I4 to I6 also finished, but I3 is yet to complete. We can move the results for I1 and I2 out of the reorder buffer into their respective architectural registers. However, I4 to I6 must wait until I3 has completed. Fig 1 illustrates the basic idea.





**Fig.1. A Reorder Buffer**

As described, the reorder buffer does not solve our problems - though it does solve another one (see below). However, even though the results of some instructions (I5 and I6 above) cannot be moved to their architectural destinations, they can still be used in computations. Suppose, in the example above that we execute a further instruction I7, which uses some register R2 as an operand. Suppose further that the result of I5 will eventually be stored in R2, and this is the actual value required by I7. Even though the value in the reorder buffer computed by I5 has not yet been moved to R2, we can still use it in the computation of I7. This process is called *forwarding* or *bypassing*, and we have mentioned it already when we considered basic pipelining - though not seen how to implement it. The reorder buffer effectively provides the history mechanism required for register renaming. The oldest version of a register is that stored in the architectural register; the next oldest is that nearest the head of the reorder buffer; the youngest is that nearest the tail of the reorder buffer.

(In practice of course, the details of implementation are not that straightforward. Reorder buffer entries need to store considerable amounts of information about instruction results - the instruction, its eventual destination, whether the result is valid or not - and it is important to be able to access all this information quickly. In order to avoid stalling the pipeline, we must be able to quickly identify when a result - needed by another instruction - becomes available, and also fetch it quickly.)

**8.2.1. Maintaining a Precise Architectural States**

The other problem that reorder buffers solve is that of maintaining a *precise architectural state*. Recall from an earlier chapter the problem of an instruction  $i+1$  terminating before instruction  $i$ , and then  $i$  causing an error. Reorder buffers solve this by effectively keeping instruction results provisional until earlier ones are known - provided instructions are actually issued in program order. Suppose in the example above that instruction I3 caused an error. We simply discard the contents of the reorder buffer (including the already executed instructions I4 to I6) and restart execution at I3 (if possible). We may waste some work (redoing I4 to I6), but we maintain backward compatibility, and a precise architectural state (because only the results of I1 and I2 will have been written out to architectural registers). Note that this is only actually the case if we *issue* the instructions in program order, and hence the order of the entries in the reorder buffer reflect program order. However, recall that we generally do issue instructions in order.

A final point to note is that reorder buffers do not cause any additional workload when we do a context switch, since the contents of the reorder buffer do *not* have to be saved. We do of course have to wait for any outstanding instructions to leave the buffer (or just dump the results and reexecute the dumped instructions), but this is much quicker than the (slow) process of saving registers to memory.

**8.3. Another Register Renaming Strategy**

Reorder buffers are convenient and simple (at least conceptually). They are also widely used (for example, all P6-based processors (Pentium Pro, Pentium II and all Pentium III-series processors use reorder buffers). However, they are not completely without disadvantages. For example, they add an extra step to the pipeline - moving results from reorder buffer to architectural registers. What is more, there is generally a limit on the number of entries that can be moved simultaneously. For example, suppose there are six execution units. In principle, this means six instructions can complete simultaneously. In practice, this will not happen often - it is unlikely we will feel it is worthwhile designing a reorder buffer and register set that will allow six results to be moved at once. So when it does happen, or whenever more instructions complete than we can move simultaneously, the pipeline

will stall. (Actually it may not all stall - only the execution unit(s) that wait.) Also, a reorder buffer is an additional place that execution units (or issue units) must look for operands in addition to the actual registers, and generally somewhere else as well (as we will see when we look at algorithms). This again means more work to do, and potentially worse performance.

An alternative is instead to provide a large set of registers and *dynamically* decide at any point in time which ones represent the actual architectural registers. For example, in a machine with 16 architectural registers we might provide a set of 64 physical registers. At any point in time, only 16 of these will correspond with the actual architectural register set - precisely *which* 16 changing as the program executes. This method - usually just called Register Renaming - solves the problems above. However, it has problems of its own. One is deciding at any one time which registers are *live* - that is, contain results that are still needed. Note that the live registers are *not* exactly the same set of registers as those corresponding with the architectural ones - determining those is another problem. Most of the time it does not matter which registers correspond with the architectural ones. However, when a context switch occurs it is necessary to know which ones to save - so there must be a means of finding this out. This is potentially a fair amount of work - however much of it can be done in parallel with the operation of the rest of the pipeline, meaning it is not on the critical path and will not slow it down. The Pentium 4 has chosen to change from a reorder buffer to register renaming.

[Previous](#) [Contents](#) [Next](#)